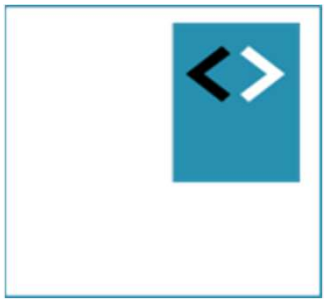
Angular Advanced RxJS and Operators

Peter Kassenaar –
info@kassenaar.com

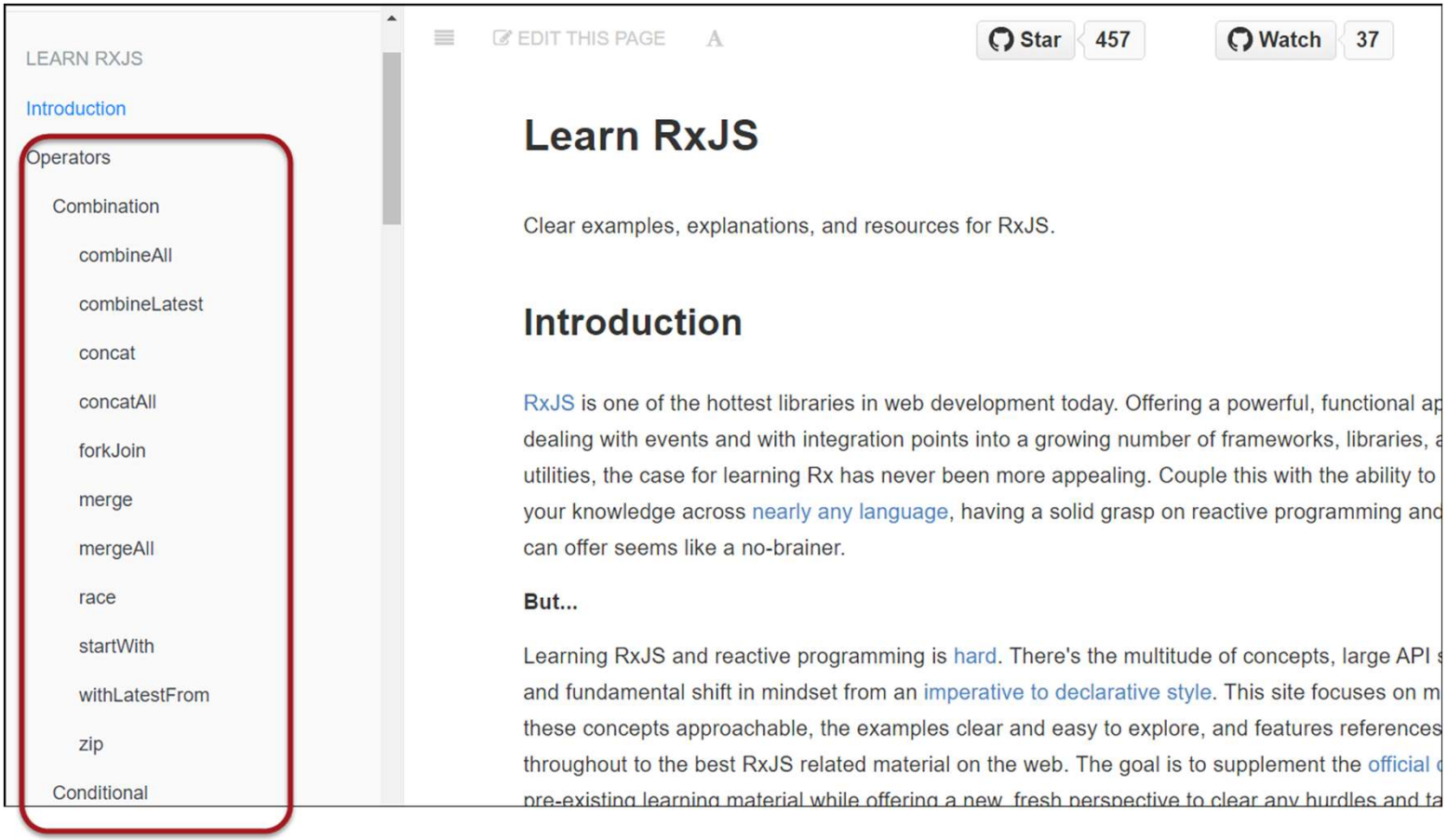


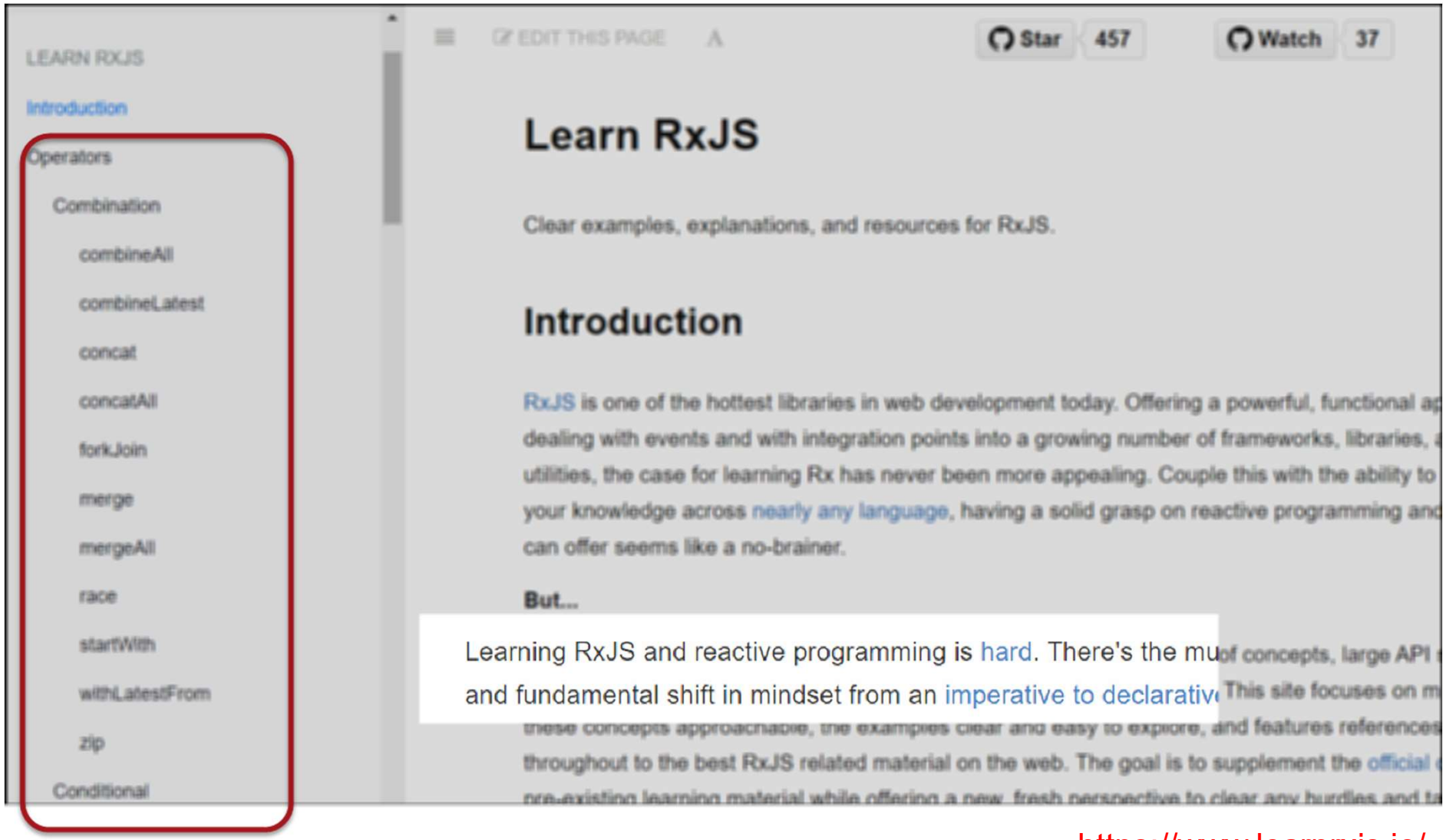
More on operators

Transforming your stream in the `.pipe()` function

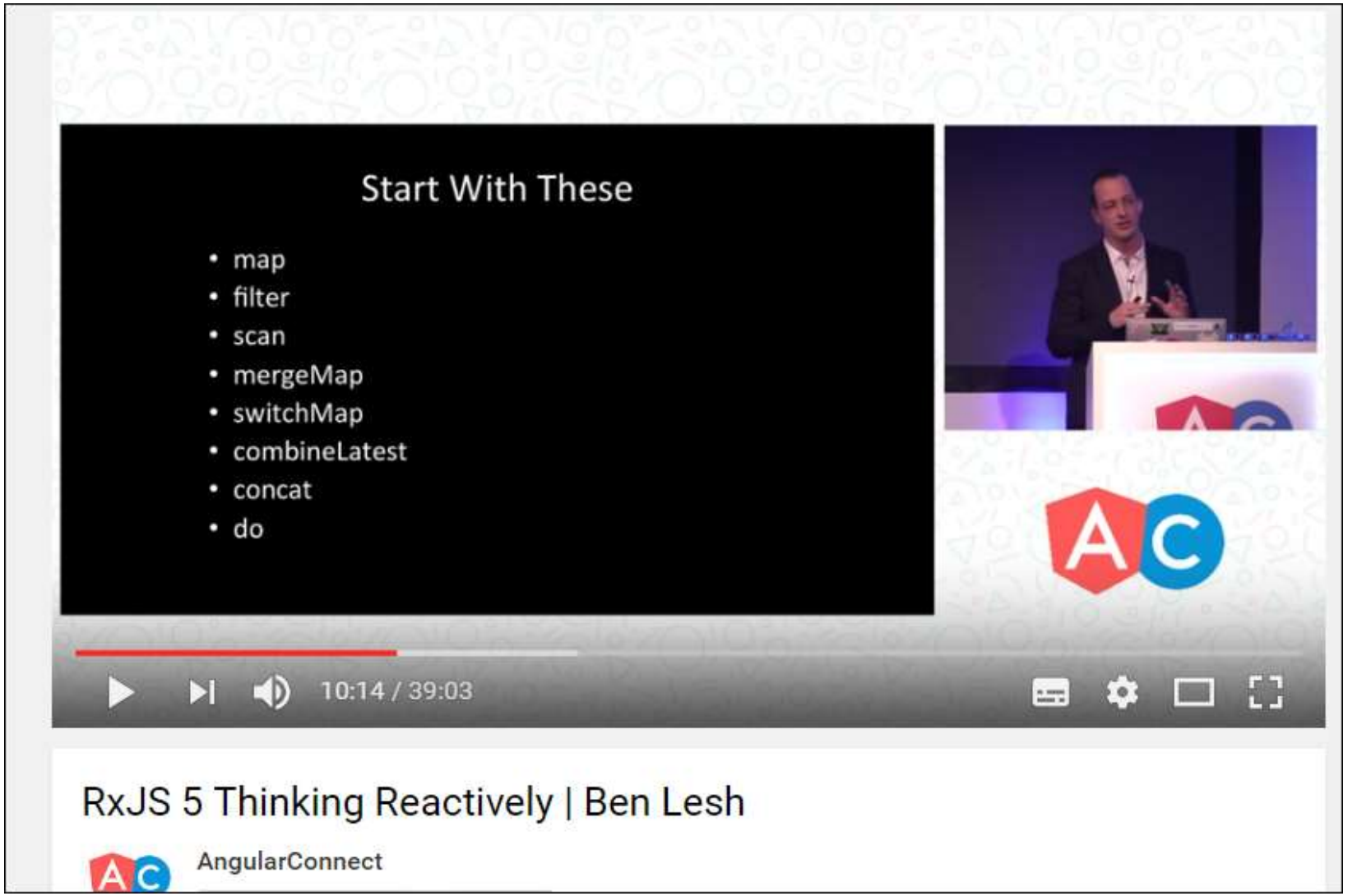
The Problem with RxJS Operators...

- There are so many of them...





https://www.learnrxjs.io/

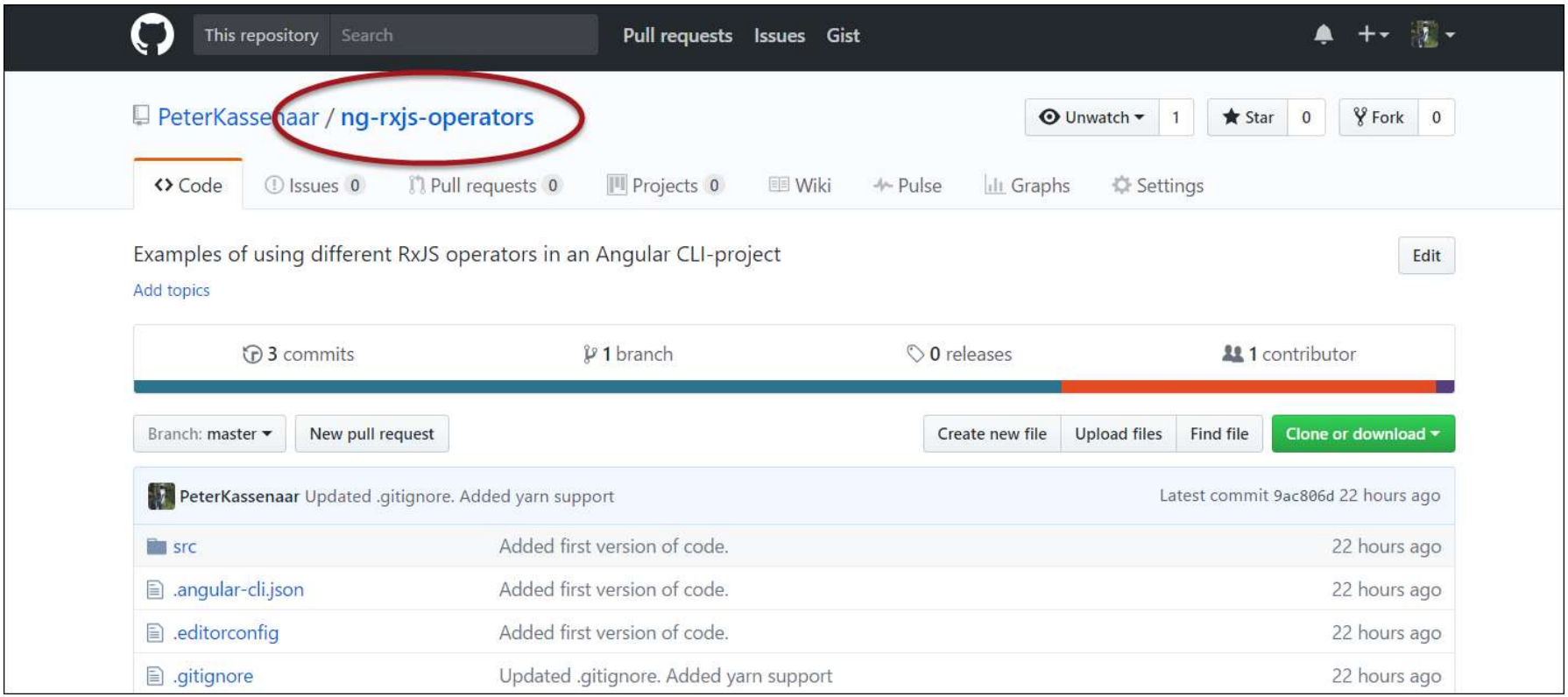


https://www.youtube.com/watch?v=3LKMwkuK0ZE

"Don't try to "Rx everything". Start with the operators you know, and go imperatively from there. There's nothing wrong with that."

- Ben Lesh

Example code



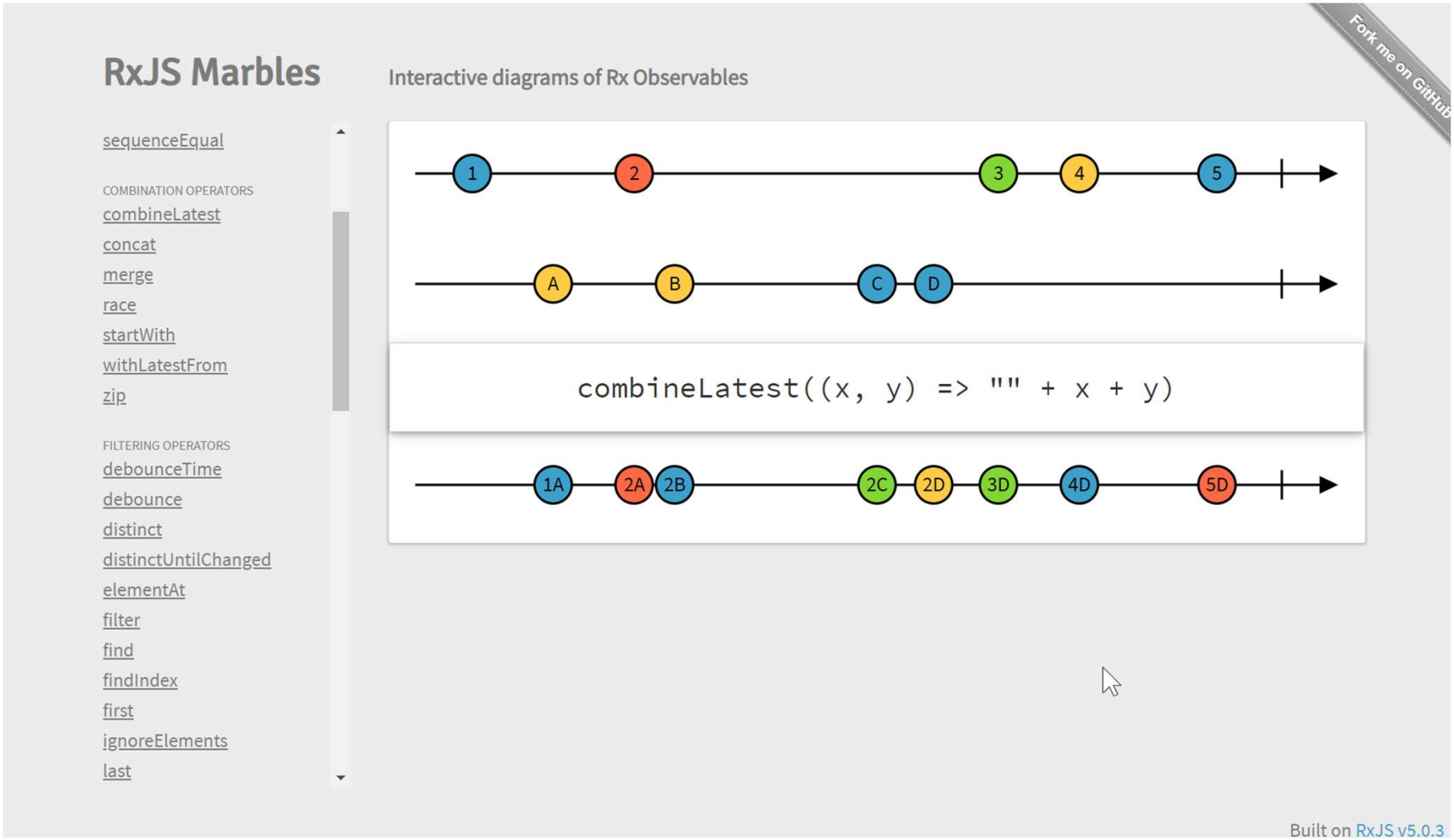
https://github.com/PeterKassenaar/ng-rxjs-operators

This presentation - 2 sections

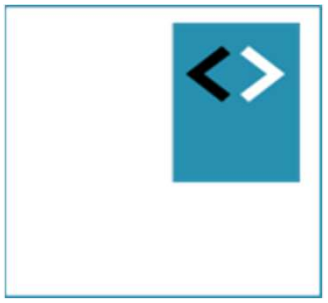
1. Basic Streams
   - Create a stream, or multiple streams from scratch, based on a UI-element
   - Subscribe to that stream(s) and do something
2. Operators

   Demo the purpose and inner workings of some often used operators

Helpful resource - rxMarbles



https://rxmarbles.com/

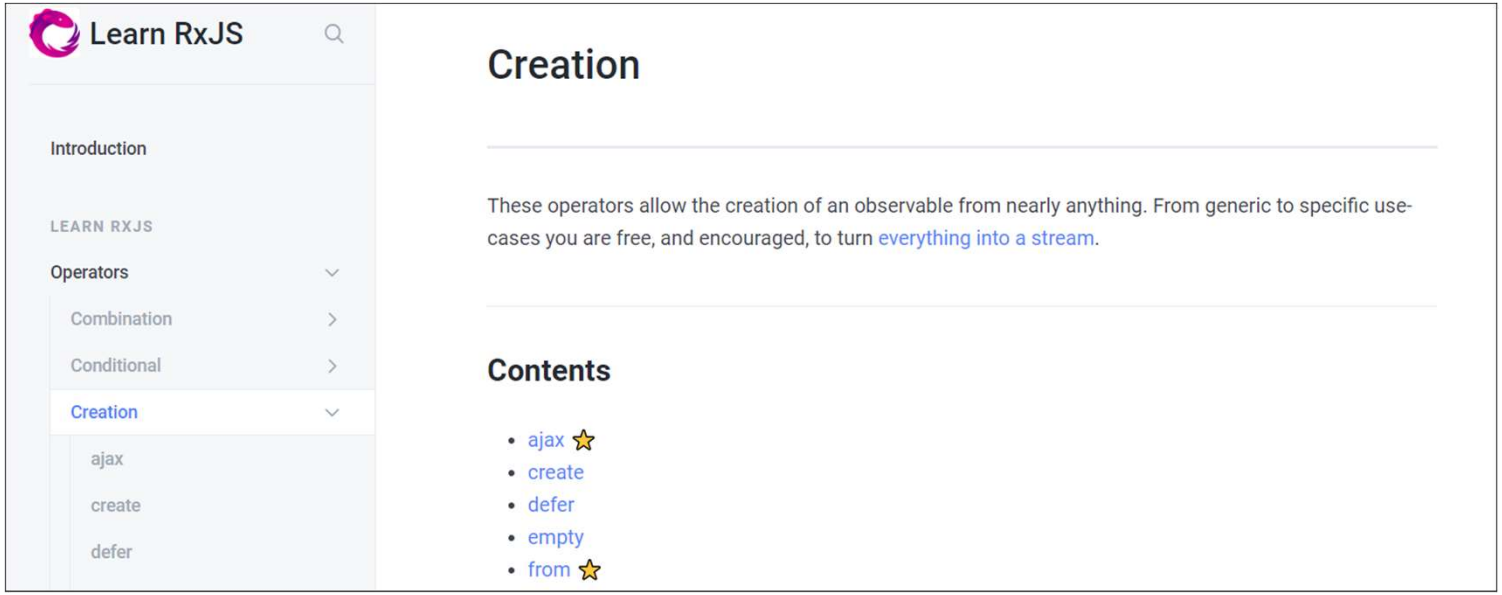


Basic streams

Creating observables from scratch

Turn something into an observable

- Basic creation operators
  - `create ()` – create an observable manually
  - `from()` – turn an array, promise or iterable into an observable
  - `fromEvent()` – turn an event into an observable
  - `of()` – turn a variable into an observable, emit it's value('s) and complete
- https://www.learnrxjs.io/learn-rxjs/operators/creation

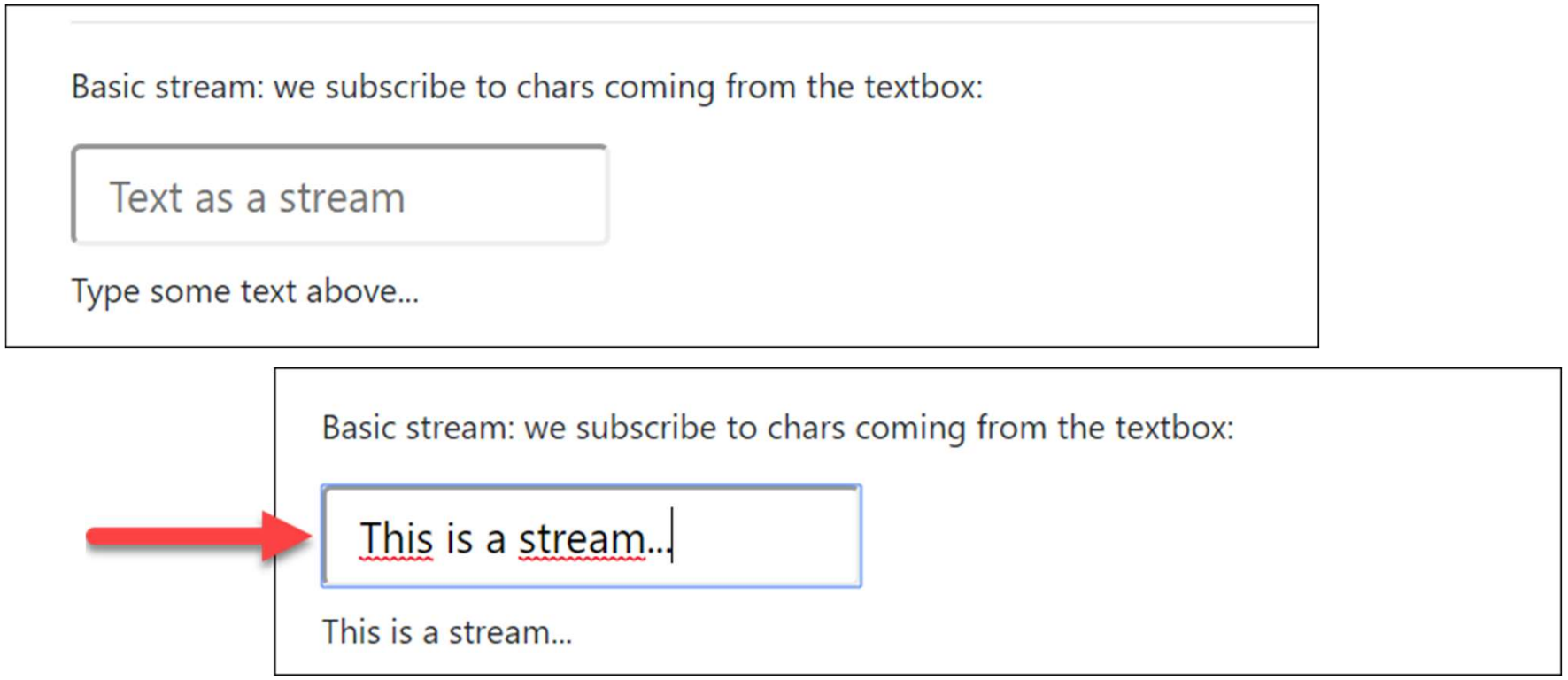


Creating an observable from a textbox

```
<input class="form-control-lg" type="text" #text1
          id="text1" placeholder="Text as a stream">
<p>{{ textStream1 }}</p>
```

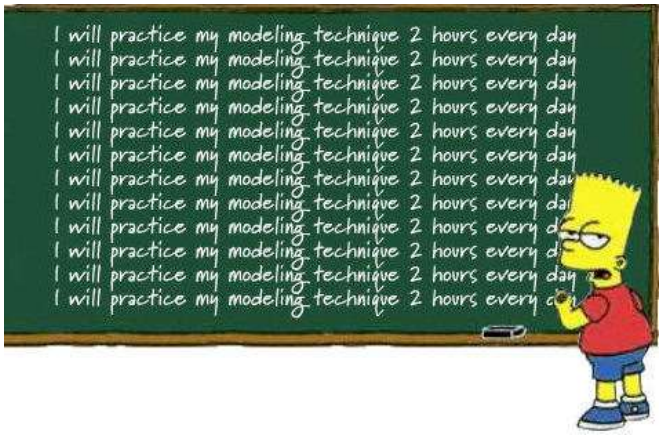
```
textStream1 = 'Type some text above...';
@ViewChild('text1', {static: true}) text1; // two parameters for @ViewChild()

// Suggestion: break the onInit() up in smaller functions
ngOnInit(): void {
  this.onTextStream1();
}

onTextStream1() {
  fromEvent(this.text1.nativeElement, 'keyup')
    .subscribe((event: any) => this.textStream1 = event.target.value);
}
```

Result


Basic stream: we subscribe to chars coming from the textbox:
Text as a stream
Type some text above...
Basic stream: we subscribe to chars coming from the textbox:
This is a stream...
This is a stream...

Workshop

- Create a textbox.
  - Add items that are typed in, to a Todo-list
  - Use an observable to subscribe to keyup-events.
- Capture key events, test if the key pressed is the Enter-key
  - If it's Enter, Add the current value of the textbox to a static array `todoItems[]`
- Example `../basic-stream/basic-stream.component.ts`
  - Start from scratch or expand this component



Using the pipe()

- We can transform the stream by adding operators to the pipeline
- Inside the pipeline we calculate new values and bind them to the UI, using generic attribute binding […]

```
<button #btnRight class="btn btn-secondary">Right</button>
Mario is moved by observable streams from the button click.
<div>
  <img id="mario" #mario
       [style.left]="position.x + 'px'"
       [style.right]="position.y + 'px'"
       src="../assets/img/mario-1.png" alt="Mario">
</div>
```

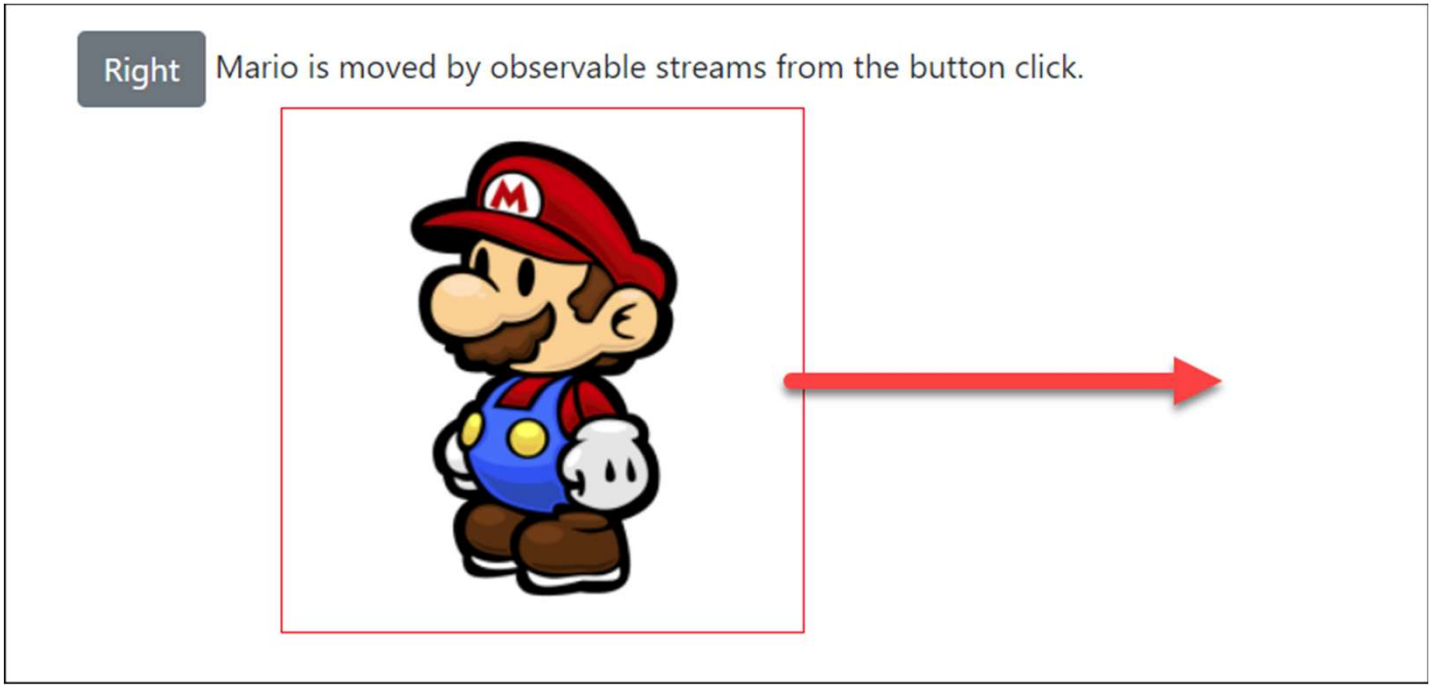
Inside our class

```
// mario
position: any;
@ViewChild('btnRight', {static: true}) btnRight;
```

```
fromEvent(this.btnRight.nativeElement, 'click')
  .pipe(
    map(event => 10), // 1. map the event to a useful value, in this case 10px
    startWith({x: 100, y: 100}), // 2. start with an object of { 100, 100}.
    scan((acc: any, current: number) => { // 3, use the scan operator as reducer function.
      return {
        x: acc.x + current,
        y: acc.y
      };
    })
  )
  .subscribe(result => {
    this.position = result;
  });
```

The result of the previous operator is the input of the next operator in the pipeline

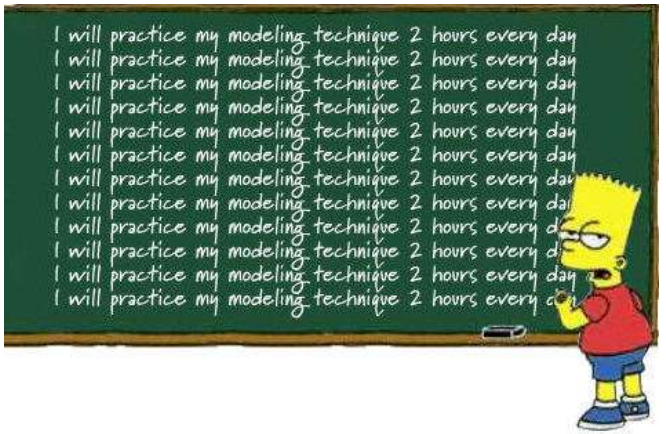
Result

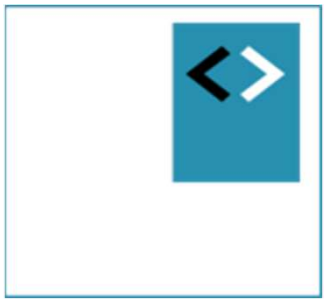

Right
Mario is moved by observable streams from the button click.

Workshop

- Create button that moves Mario to the left.
- Example `../basic-stream/basic-stream.component.ts`
  - Start from scratch or expand this component

https://github.com/PeterKassenaar/ng-rxjs-operators



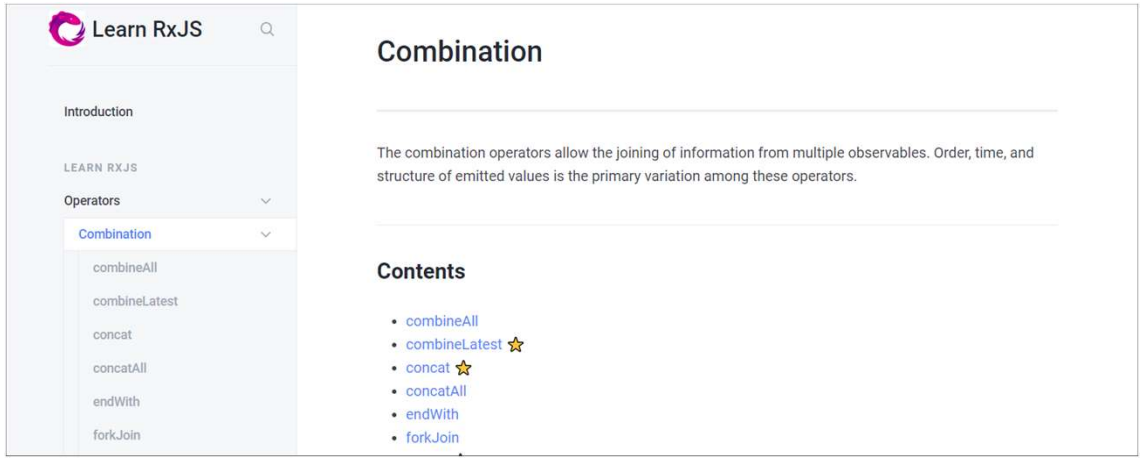


Combining streams

There are **a *lot*** of options to combine multiple streams into one stream

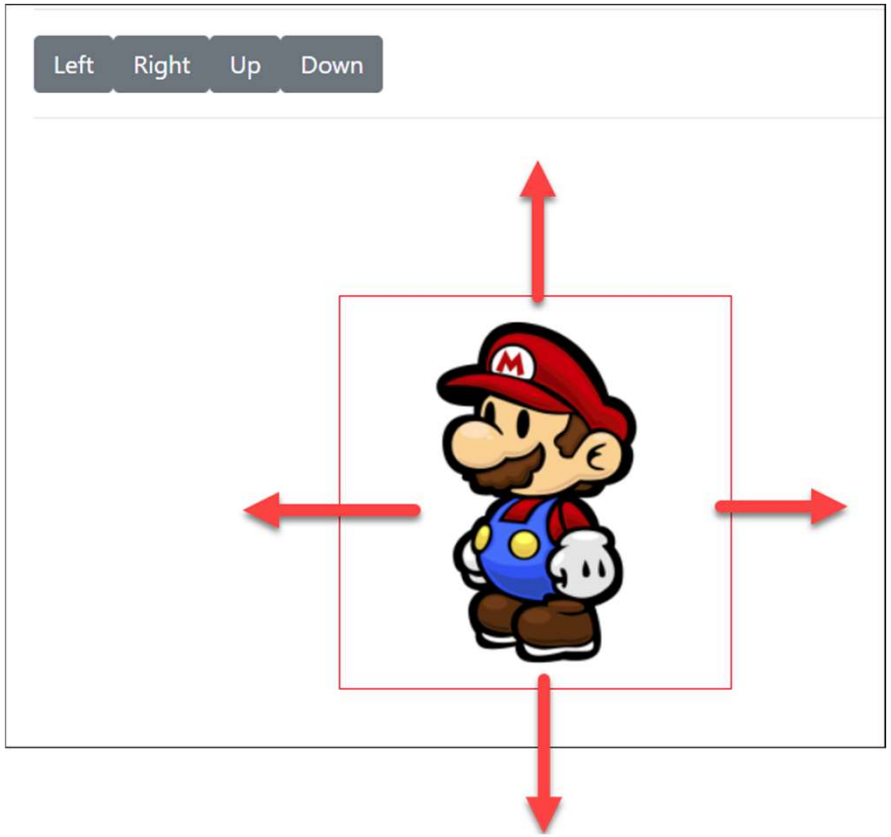
Options for combining streams

- `combineLatest()` - When any observable emits a value, emit the last emitted value from each
- `concat()` - Subscribe to observables in order as previous completes
- `forkJoin()` - When all observables complete, emit the last emitted value from each
- `merge()` - Turn multiple observables into a single observable
- `startWith()` - Emit given value first
- https://www.learnrxjs.io/learn-rxjs/operators/combination



Use case: move Mario

- Create four buttons to move Mario in different directions
  - all combined in one stream and one subscription



Template

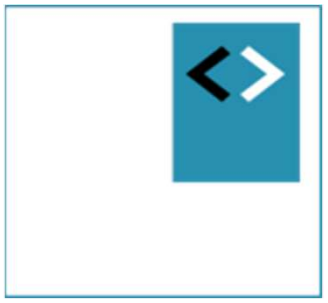
```
<button #btnLeft> Left </button>
<button #btnRight> Right </button>
<button #btnUp> Up </button>
<button #btnDown> Down </button>
<hr>

<img id="mario" #mario
     [style.left]="position.x + 'px'"
     [style.top]="position.y + 'px'"
     src="../assets/img/mario-1.png" alt="Mario">
```

```
position: any;
@ViewChild('btnLeft', {static: true}) btnLeft;
@ViewChild('btnRight', {static: true}) btnRight;
@ViewChild('btnUp', {static: true}) btnUp;
@ViewChild('btnDown', {static: true}) btnDown;

ngOnInit(): void {
  // Create multiple streams
  const right$ = fromEvent(this.btnRight.nativeElement, 'click')
    .pipe(
      map(event => {
        return {direction: 'horizontal', value: 10};
      }) // 10 px to the right
    );
  const left$ = fromEvent(this.btnLeft.nativeElement, 'click')
    .pipe(
      map(event => {
        return {direction: 'horizontal', value: -10};
      }) // -10 px to the left
    );
  …
  // combine our streams
  merge(right$, left$, up$, down$)
   .pipe(
    startWith({x: 200, y: 100}),
    scan((acc: any, current: any) => {
    return {
        x: acc.x + (current.direction === 'horizontal' ? current.value : 0),
        y: acc.y + (current.direction === 'vertical' ? current.value : 0)
      };
    })
  ).subscribe(result => {
    this.position = result;
  });
```

Create 4 different streams

Merge the streams

One subscriber



Sequencing streams

Use streams to start other streams so you can use the combined behavior

Use case: drag & drop

- We want to move Mario around with the mouse: drag-n-drop
- We compose different streams for that:
  - `down$`, when the mouse is pressed
  - `move$`, when the mouse is moved, return the current mouse position
  - `up$`, when the mouse is released
- Again, we have only one (1) subscription
  - Subscribe to the `down$`. This is the initiator
  - After the initial event, switch to the `move$`: using the `switchMap()` operator
  - But only until the `up$` event occurs!
  - Then complete the observable an release Mario
- This translates to the following code:

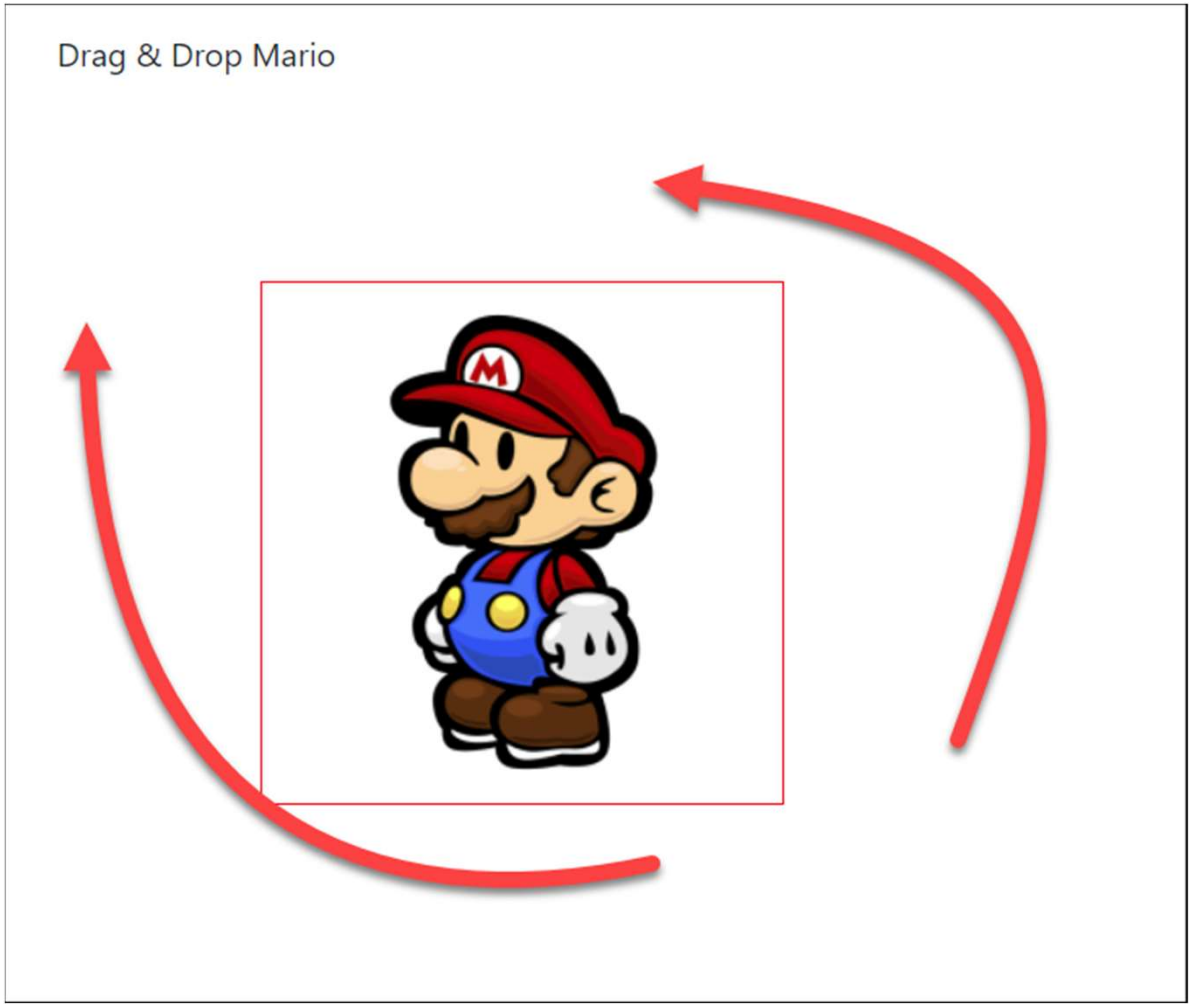
```
// correction factor for image and current page. Your mileage may vary!
  const OFFSET_X = 180;
  const OFFSET_Y = 280;

  // 1. With Drag and drop, first you capture the mousedown event
  const down$ = fromEvent(document, 'mousedown');

  // 2. What to do when the mouse moves
  const move$ = fromEvent(document, 'mousemove')
    .pipe(
      map((event: any) => {
        return {x: event.pageX - OFFSET_X, y: event.pageY - OFFSET_Y}; // OFFSET as correction factor
      })
    );

  // 3. Capture the mouseup event
  const up$ = fromEvent(document, 'mouseup');

  // 4. extend the down$ stream and subscribe
  down$
    .pipe(
      // IF the down$-event happens, we ar no longer interested in it. Instead,
      // we switch the focus to the move$ event.
      // BUT: we are only interested in the move until the mouse is released again.
      // So we add *another* pipe and use the takeUntil() operator.
      switchMap(event => move$.pipe(
        takeUntil(up$)
      )),
      startWith(this.position)
    )
    .subscribe(result => this.position = result);
}
```

Compose streams

Switch execution to another observable

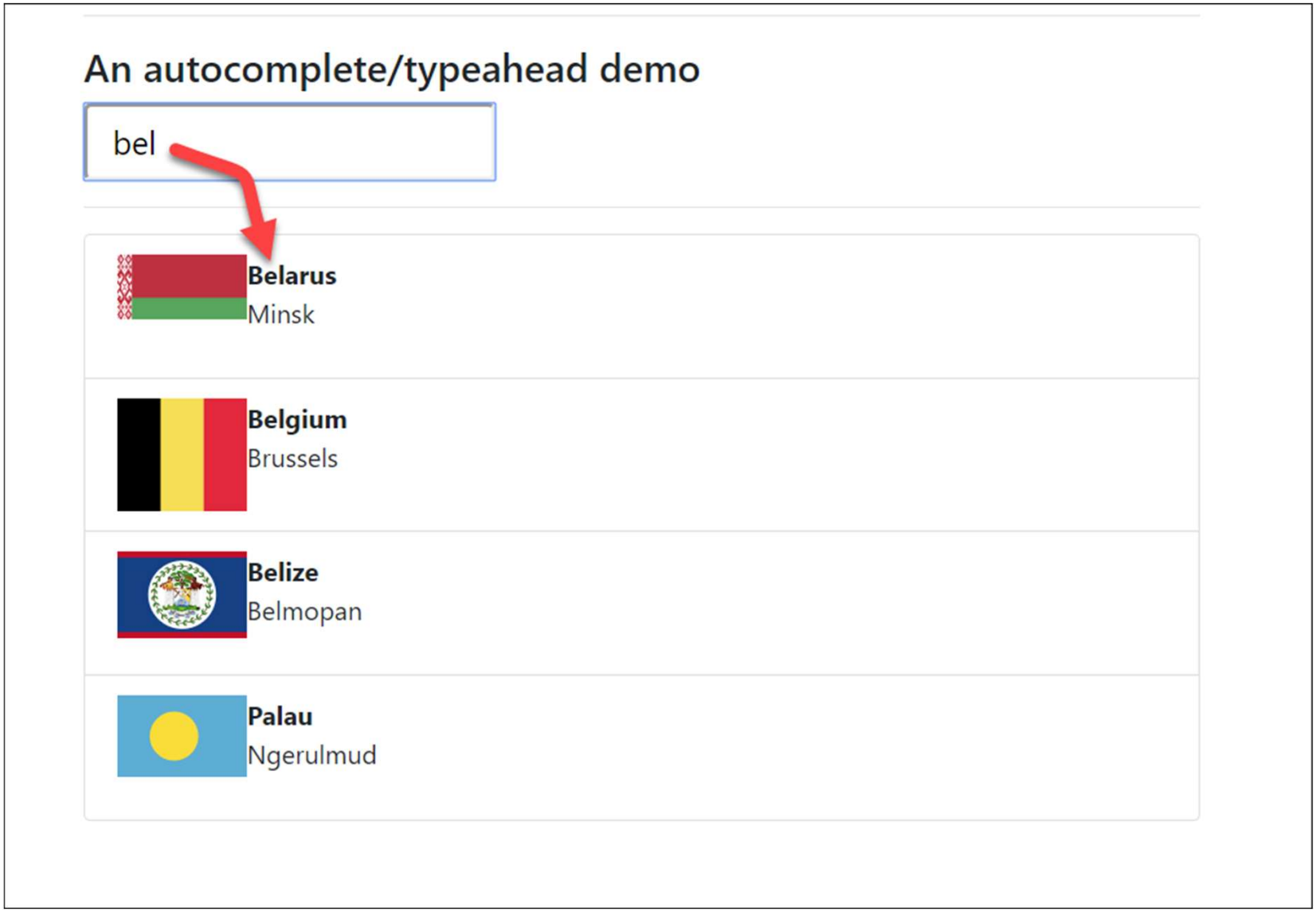
Subscribe and update position

Result



`../streams/drag-drop-stream.component.html|ts`

An autocomplete/typeahead demo



Template

As per usual – little HTML

```
<h4>An autocomplete/typeahead demo</h4>
<!--Get the keyword, the class is subscribed to this inputbox-->
<input type="text" placeholder="Country name..."
       #typeahead class="form-control-lg">
<hr/>
<!--Results-->
<ul class="list-group">
  <li class="list-group-item" *ngFor="let country of countries$ | async ">
    <img src="{{ country.flag }}" alt="" width="80" style="float: left">
    <p>
      <strong>{{ country.name}}</strong><br>
      {{ country.capital }}
    </p>
  </li>
</ul>
```

This time: using the `async` pipe

Code

```
interface ICountry {
  name: string;
  capital: string;
  flag: string;
}

const errorCountry: ICountry = {
  name: 'Error',
  capital: 'Not found',
  flag: ''
};
```

Creating interface and simple error message

```
@ViewChild('typeahead', {static: true}) typeahead;
countries$: Observable<ICountry[]>;

constructor(private http: HttpClient) {
}
```

Inside the component class

```
ngOnInit(): void {
  this.countries$ = fromEvent(this.typeahead.nativeElement, 'keyup')
    .pipe(
      filter((e: any) => e.target.value.length >= 2),
      debounceTime(400),
      map((e: any) => e.target.value),
      distinctUntilChanged(),
      switchMap(keyword => this.getCountries(keyword))
    );
}
```

Main method. See example code for comments

../streams/typeahead-stream.component.html|ts

Fetching the countries

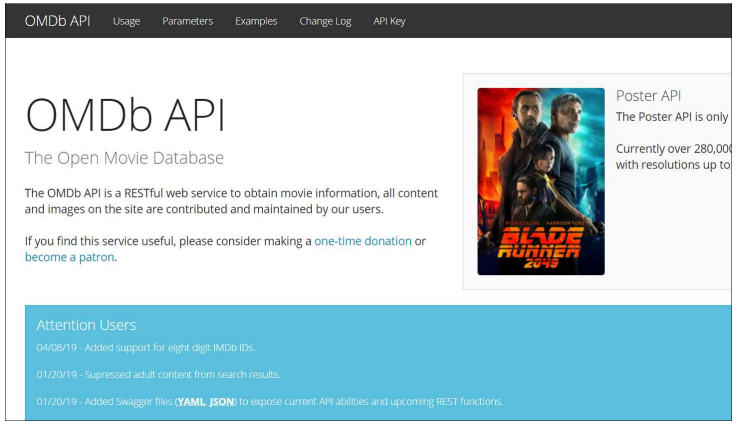
```
getCountries(keyword): Observable<ICountry[]> {
  // 7. Create the actual http-call.
   return this.http
    .get<ICountry[]>(`https://restcountries.eu/rest/v2/name/${keyword}?fields=name;capital;flag`)
    .pipe(
      // 8. catch http-errors and return a 'not found' country
      catchError(err => {
        console.log(err);
        return of([errorCountry]);
      })
    );
}
```

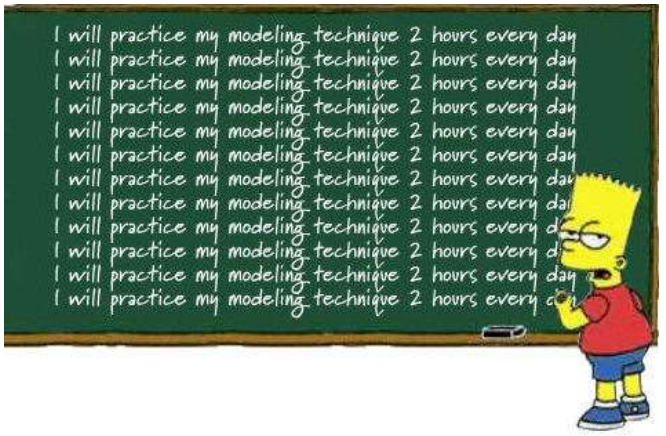
Documentation:

- filter: https://www.learnrxjs.io/learn-rxjs/operators/filtering/filter
- debounceTime: https://www.learnrxjs.io/learn-rxjs/operators/filtering/debouncetime
- distinctUntilChanged: https://www.learnrxjs.io/learn-rxjs/operators/filtering/distinctuntilchanged
- switchMap: https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap
- catchError: https://www.learnrxjs.io/learn-rxjs/operators/error_handling/catch

Workshop

- Search movies in the Open Movie Database.
  - `https://www.omdbapi.com/?apikey=f1f56c8e&s=[keyword]`
  - Create an AutoComplete inputfield for movie titels.
  - For instance, if you type `'ava…'` it should show movies like `Avatar`, and more
- Example `../basic-stream/typeahead.component.ts`
  - Start from scratch or expand this component





More operators

Getting familiar with some of the more used operators

map() and mapTo()

```
const source = from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

// .map() - apply a function every emitted output.
source.pipe(
  map((val: number) => val = val * 10)
)
  .subscribe(result => this.mapData.push(result));


// .mapTo() - map the emission to a constant value
source.pipe(
  mapTo('Hello World')
)

  .subscribe(result => this.mapToData.push(result));
```

https://rxjs-dev.firebaseapp.com/api/operators/map

filter()

```
const source       = Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
const sourceCities = Observable.from([
   'Haarlem',
   'Breda',
   'Amsterdam',
   'Groningen',
   'Hengelo',
   …
]);
```

```
// filter() - only emit values that pass the provided condition.
// In this case: only even numbers are passed through.
source.pipe(
  filter(val => val % 2 === 0)
)
  .subscribe(result => this.filterData.push(result));

// Emit only the cities that starts with an 'H'.
sourceCities.pipe(filter(city => city.startsWith('H')))
  .subscribe(result => this.cityData.push(result));
```

https://rxjs-dev.firebaseapp.com/api/operators/filter

scan()

```
const source        = Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
```

```
// .scan() acts as the classic .reduce() function on array's. It takes an
// accumulator and the current value. The accumulator is persisted over time.
source
  .pipe(
    startWith(0),
    scan((acc, curr) => acc + curr)
  )
  .subscribe(result => this.scanData.push(result));
```

https://rxjs-dev.firebaseapp.com/api/operators/scan

concat()

- Concat subscribes to observables *in order*.
- “Only when the first one completes, let me know.”
- Then move on to the next one.

- Use `concat()` when the order of your observables matters.

- Example code: simulated delay.
  - The second observable (which is finished first) only emits when the first observable completes.

app/shared/services/data.service.ts

```
// constants that are used as pointers to some json-data
const BOOKS: string   = 'assets/data/books.json';
const AUTHORS: string = 'assets/data/authors.json';
```

```
getConcatData(): Observable<any> {
  // First call. Simulate delay of 1 second
  const authors = this.http.get(AUTHORS)
    .pipe(
      delay(2000)
    );
  // Second call. Simulate delay of 2 seconds
  const books = this.http.get(BOOKS)
    .pipe(
      delay(1000)
    );

  // return the concatenated observable. It will always deliver
  // FIRST the results of the first call. No matter how long the delay.
  return concat(authors, books);
}
```

https://rxjs-dev.firebaseapp.com/api/operators/concat

merge()

- Merge combines multiple observables into one single observable.

- It emits *as soon as* it gets a result.

- Use `merge()` when order of observables is not important.

```
getMergeData(): Observable<any> {
  // First call. Simulate delay of 3 seconds, so this observable will emit last.
  const authors = this.http.get(AUTHORS)
    .pipe(
      delay(3000)
    );


  // Second call. Simulate delay of 1 seconds, so this observable will emit first.
  const books = this.http.get(BOOKS)
    .pipe(
      delay(1000)
    );


  // return the merged observable. BOOKS will be delivered first
  return merge(authors, books);
}
```

https://rxjs-dev.firebaseapp.com/api/index/function/merge

mergeMap()

- Merges the value of an inner observable into an outer observable.
- Need only the *last* value emitted? Use `.switchMap()`.
- Use this for example to retrieve results in a second observable, based on the output of a first observable

```
getMergeMapData(authorID: number = 0): Observable<any> {
  // first http-call, outer observable.
  return this.http.get(AUTHORS)
    .pipe(
      tap(response => console.log(response)),
      map((authors: any[]) => {
        console.log(authors);
        // find the correct author, using the array .find() method
        return authors.find((author: any) => author.id === authorID);
      }),
      mergeMap((author: any) => {
        if (author) {
          // second http-call, inner observable
          return this.http.get(BOOKS)
            .pipe(map((books: any[]) => {
              // filter books, bases on authorname we found earlier.
              return books.filter((book: any) => book.author === author.name);
            }));
        } else {
          // nothing found. Return empty array
          return of([]);
        }
      })
    );
}
```

https://rxjs-dev.firebaseapp.com/api/operators/mergeMap

.forkJoin()

- Make multiple (http) requests and return one combined response *once all observables are completed*.

- `.forkJoin()` returns an array with the last emitted value from each observable.

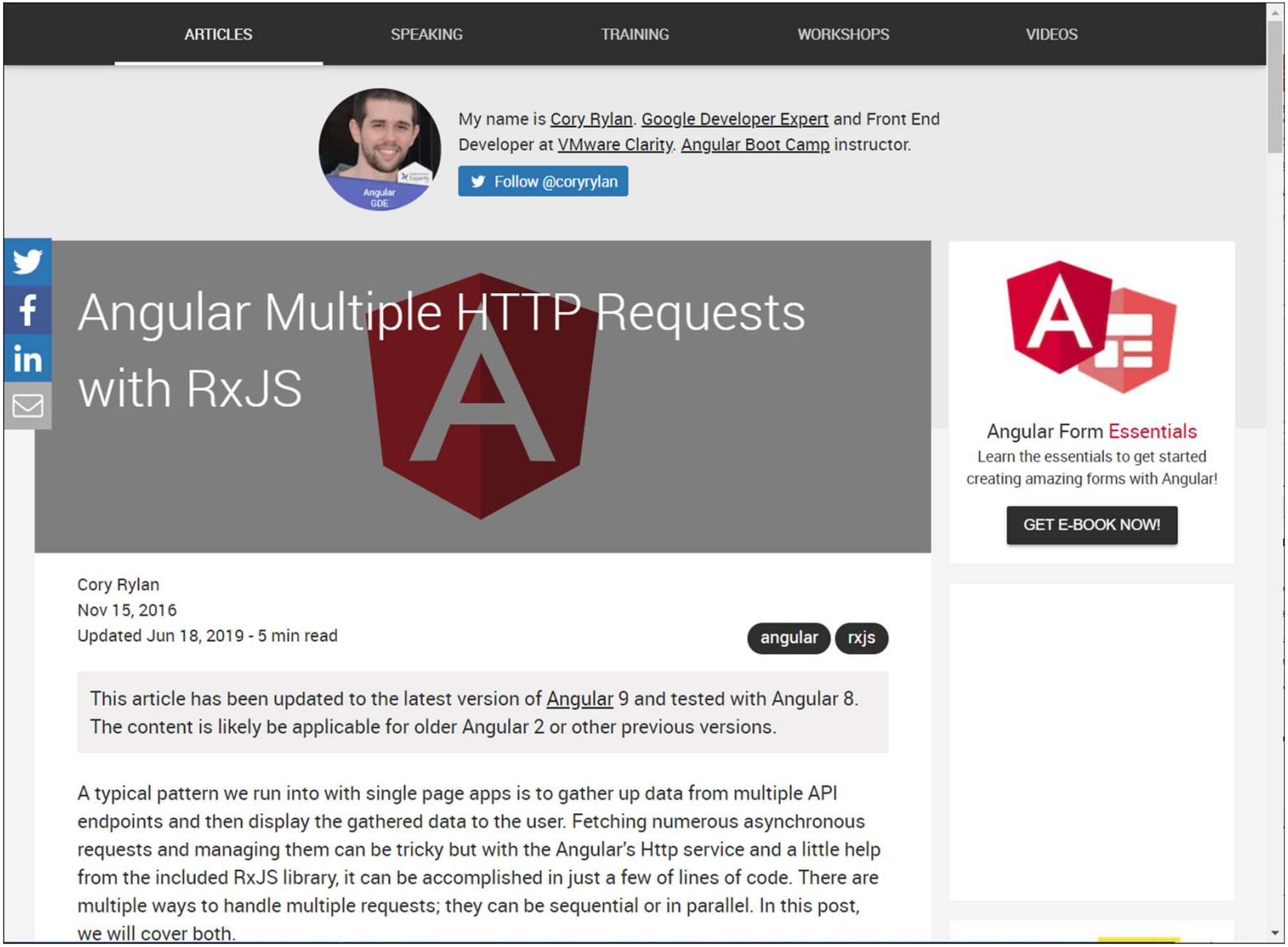
- Compose results as per your needs

```
getForkJoinData(): Observable<any> {
  return forkJoin(
    // first call
    this.http.get(AUTHORS)
      .pipe(
        delay(3000)
      ),
    // second call
    this.http.get(BOOKS)
  )
    .pipe(
      map((data: any[]) => {
        // data is now an array with 2 objects, b/c we did 2 http-calls.
        // First result, from the http-call to AUTHORS
        const author: any = data[0][0]; // Get just first author from file.
        // Second result, from the http-call to BOOKS
        const books: any[] = data[1];
        // Compose result, in this case adding the books to the extracted author.
        author.books = books.filter(book => book.author === author.name);
        return author;
      })
    );
}
```

https://rxjs-dev.firebaseapp.com/api/index/function/forkJoin

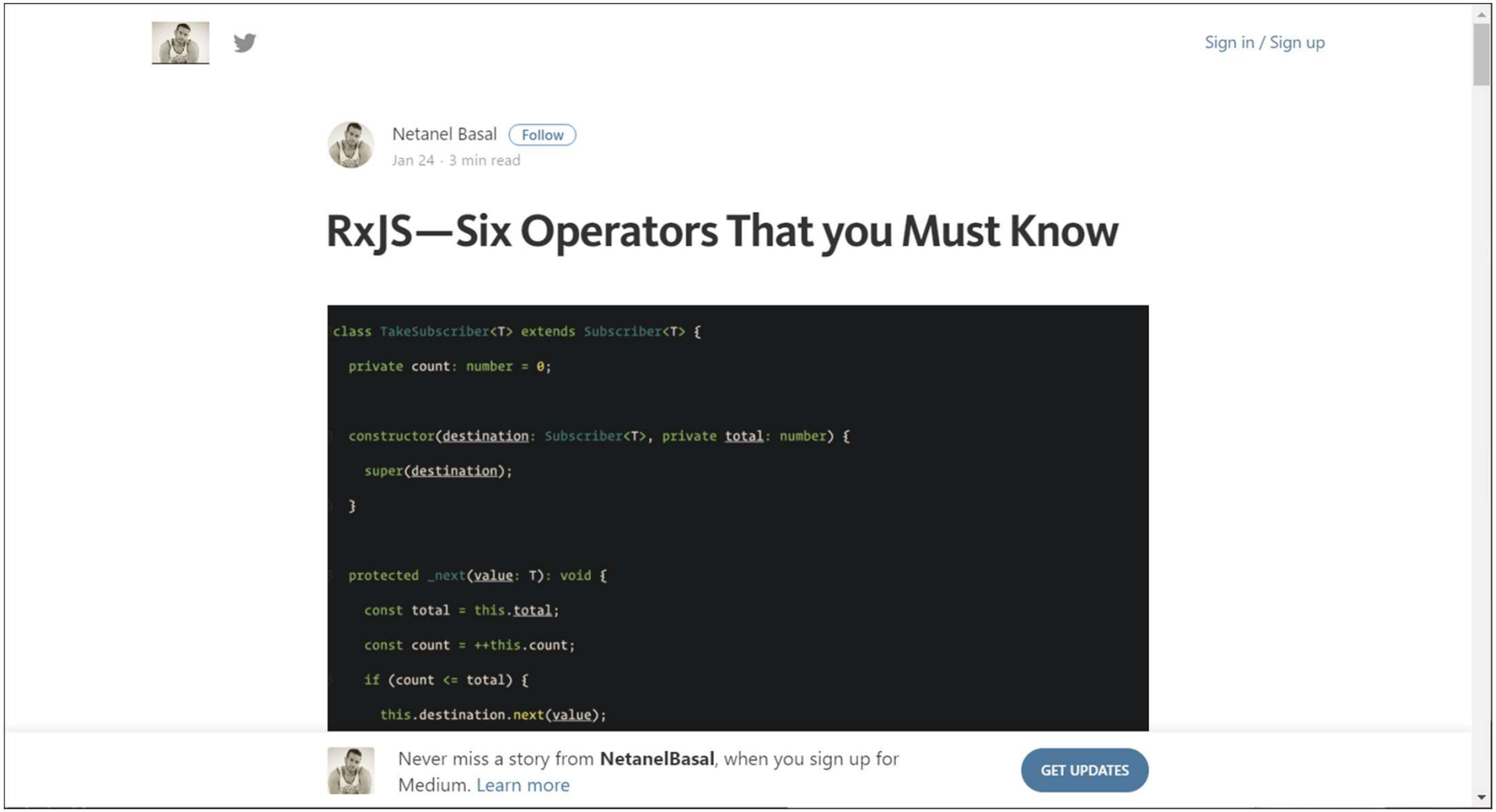
Other interesting / often used Operators

- `from()`, `of()` - create observables from almost everything
- `fromEvent()` - create observable from a DOM-event.
- `debounce()`, `debounceTime()` - Discard emitted values that take less than the specified time between output
- `tap()` - utility operator - transform side-effects, such as logging
- …and much more… See http://www.learnrxjs.io
- See also https://rxjs-dev.firebaseapp.com/

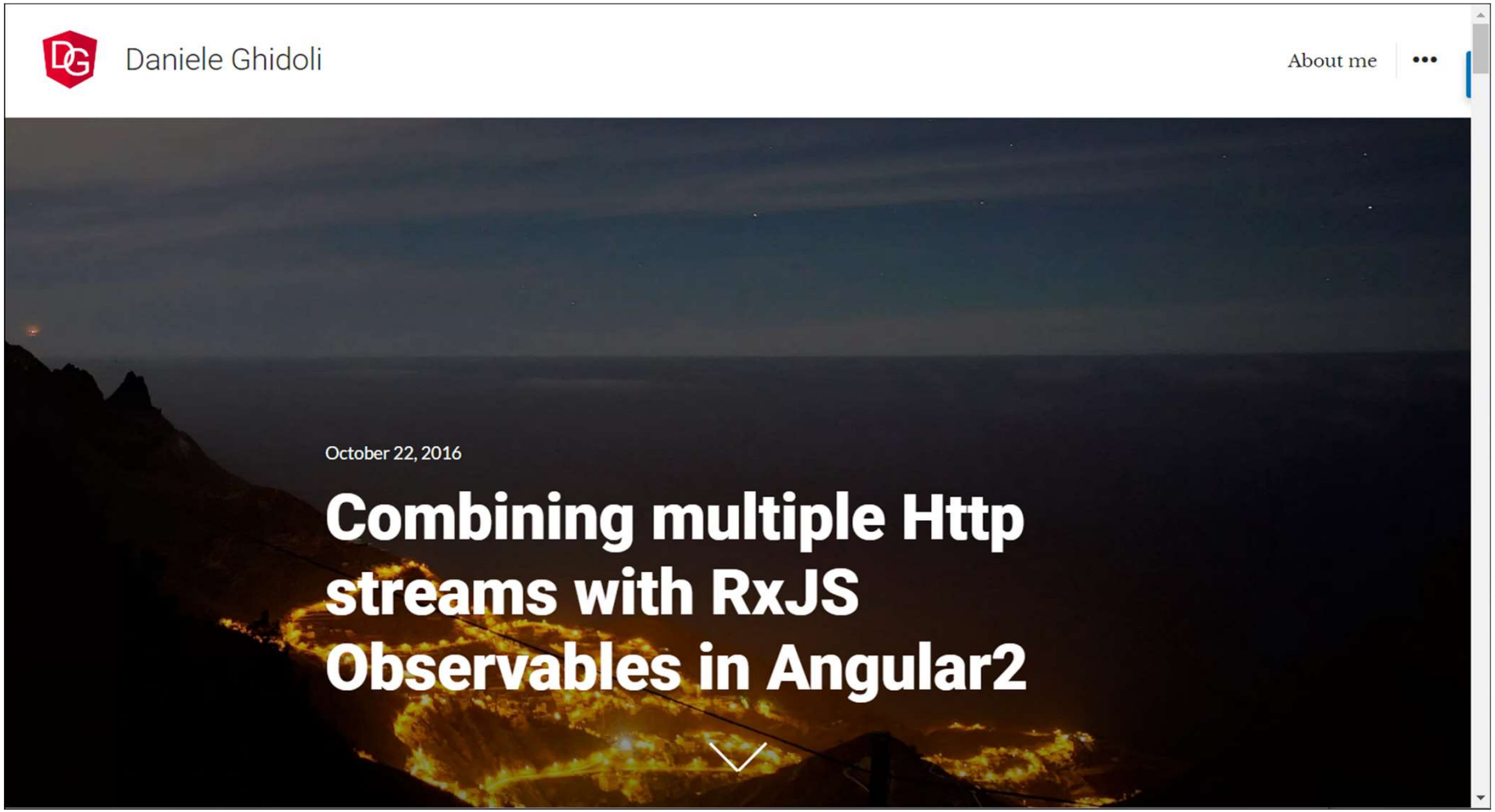


https://coryrylan.com/blog/angular-multiple-http-requests-with-rxjs

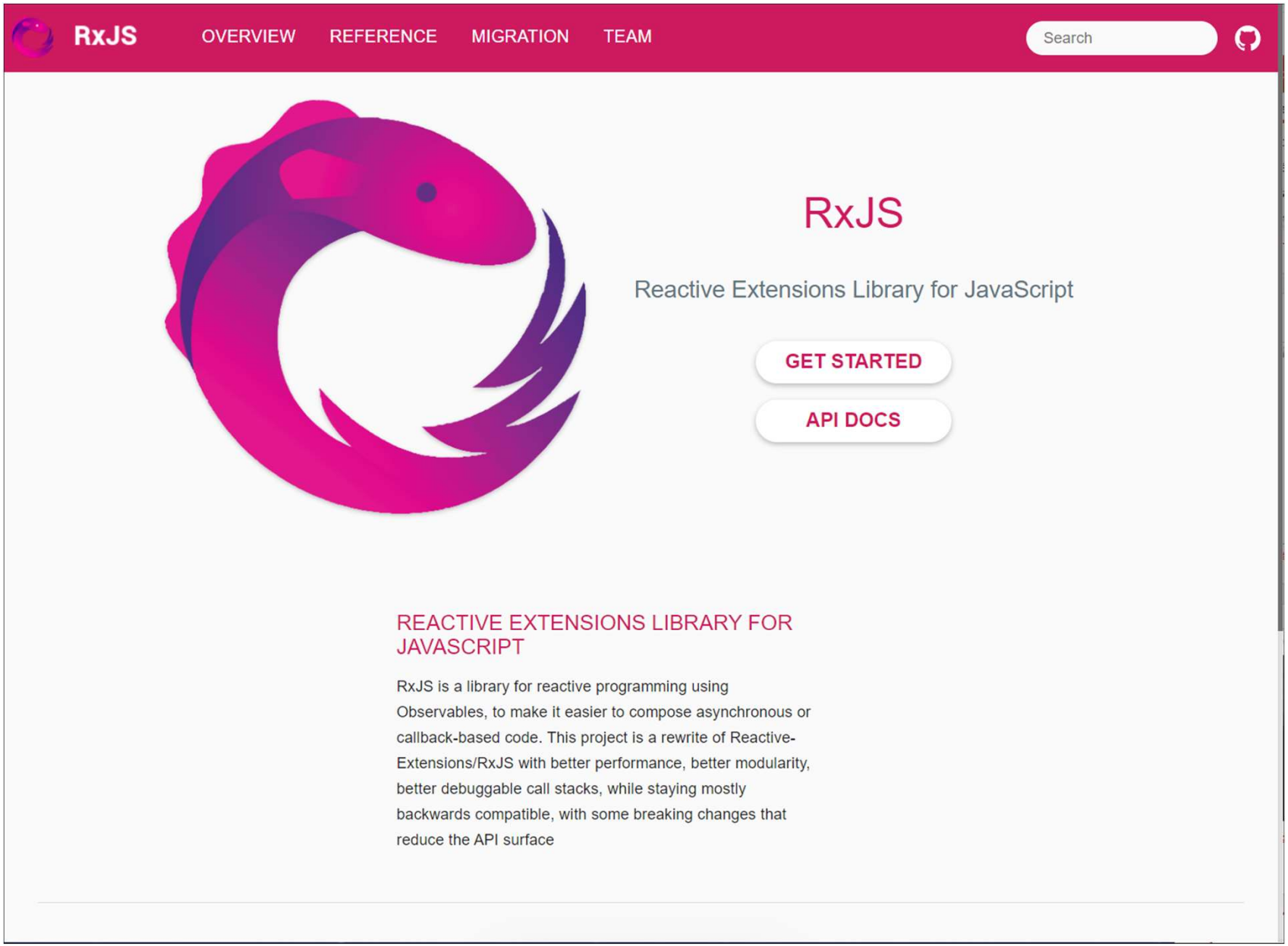
More information



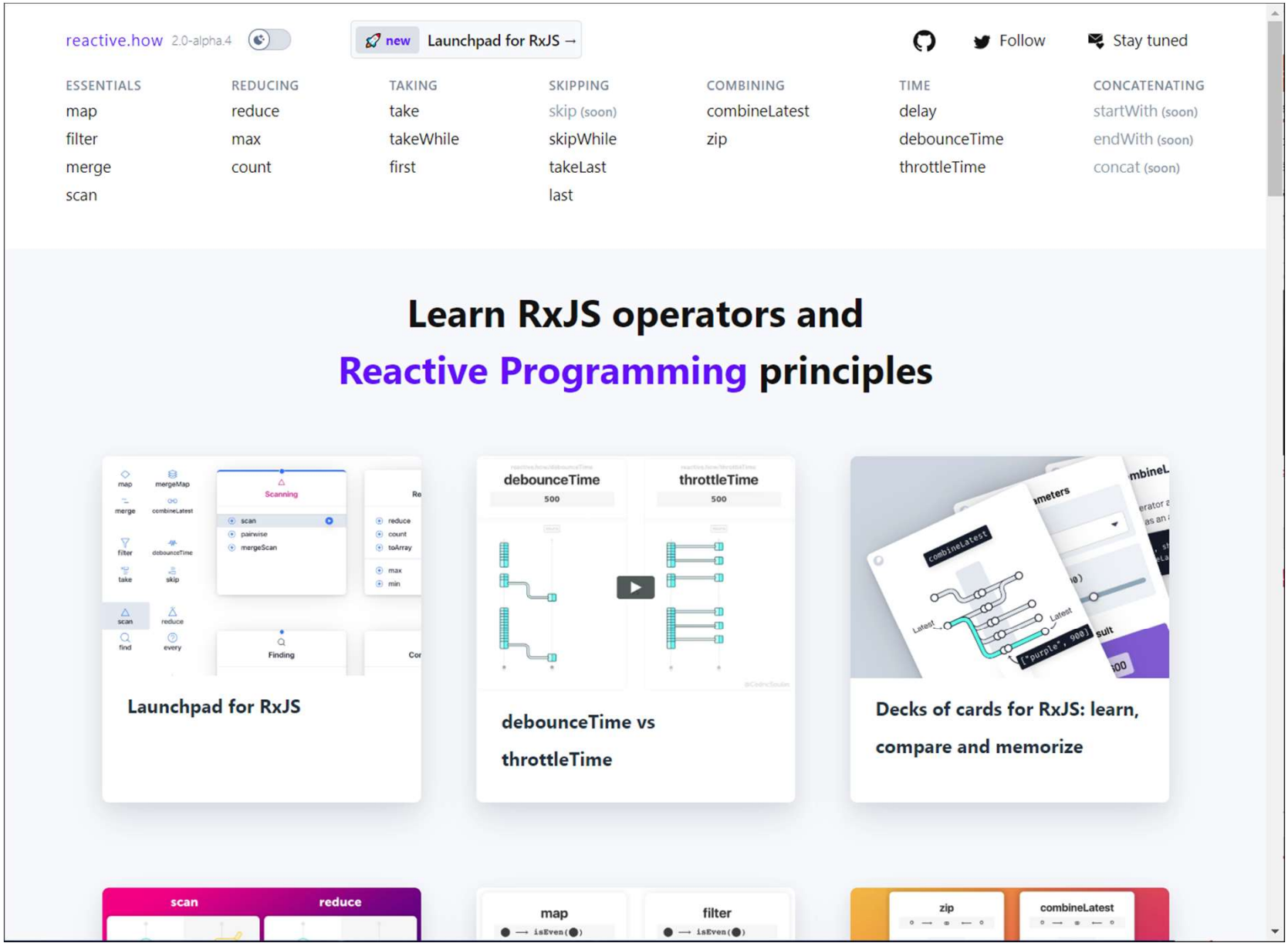
https://netbasal.com/rxjs-six-operators-that-you-must-know-5ed3b6e238a0#.ocz4xfwkl



http://blog.danieleghidoli.it/2016/10/22/http-rxjs-observables-angular/



https://rxjs-dev.firebaseapp.com/



https://reactive.how/

Workshop

- Create a call to the Open Movie Database API, using a keyword to search for 10 movies
  - http://www.omdbapi.com/?apikey=f1f56c8e&s=<keyword>
- Create an additional call to get details for every movie returned. Use the `imdbID` property for that
  - http://www.omdbapi.com/?apikey=f1f56c8e&i=<imdbID>
- Display results in the UI, first a list of movies, then details per movie as soon as they come available.
- What is your best solution? Multiple ways of doing this!